Vol.2019-IOT-44 No.27
2019/3/8

CRIUを利用したHTTPリクエスト単位でコンテナを再配置できる低コストで高速なスケジューリング手法

松本 亮介[1,a)]　近藤 宇智朗[2]

概要：スマートフォンやPCのモバイル化，SNSの爆発的普及に伴い，個人のWebサイトであってもコンテンツ次第でアクセスが集中する機会が増大してきている．我々は，サービス利用者に専門的な知識を要求せず，アクセス数や負荷に応じて反応的かつ高速にリソースをコンテナ再割当てすることで，サービス利用者や事業者に手間を強いることなく突発的なアクセス集中に耐えうるFastContainerアーキテクチャを提案した．一方で，従来のホスティングサービスやクラウドサービスと同様に，コンテナの収容サーバ障害時に，HTTPタイムアウトが生じない程度での可用性を担保しサービスを継続提供するためには，複数収容サーバに横断して，それぞれ複数コンテナを立ち上げておく必要があった．そのため，利用者にとってはサービス利用コストの増加に繋がっていた．本研究では，HTTPリクエスト処理時において，単一のコンテナであっても，収容サーバの状態に応じて自動的にコンテナを別の収容サーバに再配置しWebサービスを継続させる，HTTPリクエスト単位での低コストで高速なコンテナのスケジューリング手法を提案する．リソースコストを低減し，複数の収容サーバにコンテナを立ち上げておくことなく，高速にインスタンスを再配置するために，Webサーバソフトウェアが起動完了する直後にフックする処理を入れることでプロセスをイメージ化しておき，コンテナ再配置時にはコンテナ上のWebサーバプロセスをイメージから高速に起動させる．

Low-cost and High Performance Scheduling Method for Reactive Relocation of Containers Each HTTP Request Using CRIU

Ryosuke Matsumoto[1,a)]　Uchio Kondo[2]

1. はじめに

インターネットを活用した企業や個人の働き方の多様化に伴い，インターネット上で自らを表現する機会が増加している．特に個人にとっては，TwitterやFacebookを活用して，自身が作成したコンテンツを拡散させることにより，効率よくコンテンツへの訪問数を増やすことができるようになった．その結果，コンテンツの内容の品質が高ければ，さらに拡散され，コンテンツに紐づく個人のブランド化も可能になってきている．そのため，コンテンツ拡散時であっても継続的に配信可能なWebサイトの構築は，個人用途であっても非常に重要になってきている．

Webコンテンツを継続的に配信するためには，オートスケーリングのような負荷対策と，冗長構成による可用性の担保が重要である．一般的に，個人がWebコンテンツを配信するためには，Webホスティングサービスやクラウドサービスなどが利用される[8]．特に，個人の利用を想定した場合，単一の物理サーバに高集積にユーザ領域を収容するような低価格Webホスティングサービスが利用される．そのような低価格ホスティングサービスでは，利用者のWebコンテンツが特定のWebサーバに紐づくため，ユーザ領域単位で突発的なアクセス集中に対して負荷分散することが難しい．クラウドサービスの場合は，利用者が

[1] さくらインターネット株式会社 さくらインターネット研究所
SAKURA Research Center, SAKURA Internet Inc., Akasaka, Chuo-ku, Fukuoka 810-0042 Japan
[2] GMO ペパボ株式会社
GMO Pepabo, Inc., Tenjin, Chuo ku, Fukuoka 810-0001 Japan
[a)] r-matsumoto@sakura.ad.jp

アクセス集中に耐えうるオートスケーリング [5] の仕組みを作る必要があったり，各サービス・プロバイダが提供しているオートスケーリングの機能 [1] を使う必要がある．しかし，クラウドサービスに関する専門的な知識を有さない個人が，費用を最低限に抑えつつ，突発的なアクセス集中を予測的かつ安定的に対応することは依然として困難である．

可用性の面では，Web ホスティングサービスの方式もクラウドサービスも，仮想的に構築されたサーバ環境であるインスタンスを複数のハードウェア上に起動させることにより，特定のハードウェアに障害が置きてもサービスを停止せずに提供可能である [12]．しかし，複数台のインスタンスが必要であることから，インスタンスの利用コストが増加する．さらに，可用性のためのインスタンスをコストの兼ね合いから負荷分散用途に転用してしまうことにより，収容サーバが停止した際に縮退状態のインスタンス数でサービスを提供することになり，場合によってはリソース不足に陥る．

我々はアクセス数や負荷に応じて反応的かつ高速にリソースをインスタンス[*1]再割当てすることで，サービス利用者や事業者に手間を強いることなく突発的なアクセス集中に耐えうる FastContainer アーキテクチャ [13] を提案した．一方，可用性の面では，依然として単一のインスタンスの場合，そのインスタンスが収容されている物理サーバが正常に稼働していることが前提であり，収容サーバが停止すると，手動によるインスタンスの再配置が必要であった．そのため，障害時に HTTP タイムアウトを生じさせない程度の可用性を担保するには，従来手法と同様にインスタンスを複数の収容サーバに起動できるようにしておく必要があり，利用者にとっては利用コストの増加に繋がっていた．また，新たなインスタンスを新規で再配置する場合は，インスタンス起動時の Web サーバプロセスの起動時間がオーバーヘッドとなってレスポンスが一時的に遅くなり，ユーザ体験を損なう課題があった [10]．

プロセスの起動時間を高速化するために，CRIU[2] と呼ばれる Linux プロセスの Checkpoint/Restore 技術によってプロセスをイメージ化することにより，高速にプロセスの状態を復帰させる手法がある．しかし，Web サーバが起動してからリクエストを処理している状態のプロセスをイメージ化すると，起動後のサーバプロセスの状態やコンテンツの状態等を保持してしまうため，実用上扱いにくい．

本研究では，HTTP リクエスト処理時において，単一のインスタンスであっても，収容サーバの状態に応じて，自動的にインスタンスを別の収容サーバに再配置し，サービスを継続させる，HTTP リクエスト単位でのインスタンスのスケジューリング手法を提案する．提案手法では，FastContainer の状態変化の高速性に基づいて，インスタンスが頻繁に収容サーバを変更されてもサービスに影響がないことを利用する．それによって，プロキシサーバから収容サーバに 1 個の ICMP あるいは TCP パケットで応答速度を計測し，少ないパケット数と短いタイムアウトで収容サーバの反応時間を計測できる．．そのことで，HTTP リクエスト単位でのインスタンススケジューリングを実現する．さらに，高速にインスタンスを再配置するために，Web サーバソフトウェア自体を拡張することなく Web サーバプロセス起動完了直後にフックする処理を入れることでプロセスイメージを作成しておくことにより，高速に Web サーバプロセスをそのイメージから起動させる．

本論文の構成を述べる．2 章では，Web ホスティングサービスやクラウドサービスにおける可用性の担保のための取り組みとその課題について述べる．3 章では，2 章で述べたインスタンスの再配置の課題を解決するための提案手法のアーキテクチャおよび実装を述べる．4 章では，HTTP リクエスト単位でのインスタンススケジューリング手法の評価を行い，5 章でまとめとする．

2. Web サービス基盤の可用性

Web サイトが収容されている物理サーバに障害があった場合に，いかにサービスを継続するかといった可用性の観点でシステムを構築する必要がある．本章では，相互に関連の深い負荷分散と可用性の観点から，Web ホスティングサービスやクラウドサービスにおける関連研究と課題について整理する．

本論文における可用性とは，インスタンスの収容サーバが停止しても，クライアントからインスタンスに対するリクエストが HTTP タイムアウトせずに，オンラインでサービス継続を可能とする程度を前提とする．

2.1 Web ホスティングサービス

Web ホスティングサービスとは，複数のホストで物理サーバのリソースを共有し，それぞれの管理者のドメインに対して HTTP サーバ機能を提供するサービスである．Web ホスティングサービス [8] において，ドメイン名 (FQDN) によって識別され，対応するコンテンツを配信する機能をホストと呼ぶ．

一般的に，個人が Web コンテンツを配信するためには，Web ホスティングサービスやクラウドサービスなどが利用される [8]．特に，個人の利用を想定した場合，仮想ホスト方式を利用して，単一の物理サーバに高集積にユーザ領域を収容するような低価格 Web ホスティングサービスが利用される [11]．仮想ホスト方式では複数のホストを Apache httpd のような単一のサーバプロセス群で処理するため，

[*1] FastContainer におけるインスタンスとはリクエストを契機に一定期間起動するものであり，リクエストが送られてきていない場合は停止している場合もありうる論理的なインスタンスである．これを FastContainer では Mortal と名付ける．

リクエストは Web サーバプロセスを共有して処理される．そのため，ホスト単位で使用するリソースを適切に制御したり，その原因を迅速に調査することが困難である [14]．

Web ホスティングサービスでは，サービス利用者の Web コンテンツは特定の Web サーバに収容され，Web サーバと Web コンテンツが紐づくため，負荷に応じたオートスケーリング*2は，データの整合性の面と前述したホスト単位での適切な負荷計測・制御の面から困難である．

Web ホスティングサービスにおける負荷分散と可用性を両立する手法として，ユーザデータ領域を共有ストレージにまとめた上で，仮想ホスト方式を採用した複数台の Web サーバで負荷分散と可用性を担保する手法 [12] がある．しかし，ホストの収容サーバを複数台配置して全ホストを複数の物理サーバに配置することが前提となる手法であるため，ホスト単位でのスケーリングや可用性の担保は対応できず，コストも高くなる．また，負荷に応じて，ホスト単位での即時性の高いスケールアップ型の負荷対応もできない．

2.2 クラウドサービス

クラウドコンピューティング [7] とは，ネットワークやサーバといったコンピュータリソースのプールから必要な時に必要な量だけオンデマンドに利用可能とするコンピューティングモデルである．クラウドサービスはクラウドコンピューティングを各種サービスとして提供するサービスである．

クラウドサービスでは，Web コンテンツを配置するだけでなく，Web サーバソフトウェアやデータベースをサービス利用者が自ら構築する必要がある．そのため，負荷分散のためのシステム設計をホスト単位で個別に行うことができる点において自由度は高いが，前提として専門的な知識が必要となる．

クラウドサービスで可用性を担保するためには，複数のインスタンスを複数の収容サーバにホットスタンバイ状態で起動させておき，特定の収容サーバに障害が発生した場合には，ホットスタンバイ状態のインスタンスにリクエストが送信されるように変更する．これによって，障害対応時に，新規でインスタンスを立ち上げ直す処理を省略し，継続的にサービスを提供し続けることが可能となる．しかし，複数台のインスタンスが必要であることに起因して，必要なハードウェアコストが増加する．また，そのようなハードウェアコストを限られた予算の中で活用するために，ホットスタンバイ状態のインスタンスを負荷分散に利用することもある．しかし，収容サーバが停止すると縮退状態のインスタンスでサービス提供することになり，場合によってはリソース不足に陥る．

*2 自動的にハードウェアリソースを追加し負荷分散を実現する機能

高負荷状態に対して迅速に対応するために，事前にある程度想定される量の仮想マシンを予測的に起動させておくことによって対処する手法がある [15]．しかし，本手法は突発的なアクセスに対するスケーリングを対象としておらず，予測の精度を保つための各種パラメータの選定に関する課題もある．また，可用性を担保することは想定されていない．

上記のような問題を解決するために，クラウドサービスプロバイダの AWS は，プロバイダ指定の記法によってアプリケーションを実装すれば，自動的にコンピュータリソースを決定し，可用性を担保しながらも高負荷時には自動的にプロバイダ側でオートスケーリングする機能 [1] を提供している．しかし，前提としてプログラミングができるエンジニアを対象としており，一般的な OSS として公開されている Web アプリケーションを利用できないことが多く，個人が Web コンテンツを配信する用途においては使用上の制限が大きい．

2.3 FastContainer アーキテクチャ

松本らは，個人が Web コンテンツを配信する用途において，Web ホスティングサービスやクラウドサービスにおける突発的にアクセス集中の対応やオートスケーリングの課題を解決するために，FastContainer アーキテクチャを提案している [13]．FastContainer は，Web ホスティングサービスを利用できる程度の知識を持った個人が，WordPress のような一般的な CMS を利用した Web コンテンツを配信することを前提にしている．そのような個人のサービス利用者が負荷分散のシステム構築やライブラリの運用・管理を極力必要とせず，HTTP リクエスト単位でインスタンスの状態を決定し，迅速にユーザ領域を複数の物理サーバに展開可能で，実行環境の変化に素早く適応できる恒常性を持つシステムアーキテクチャである．

FastContainer アーキテクチャでは，インスタンスとして，仮想マシンではなく Linux コンテナ [4] を利用する．Linux コンテナはカーネルを共有しながらプロセスレベルで仮想的に OS 環境を隔離する仮想化技術のひとつである．そのため，コンテナの起動処理は仮想マシンのようなカーネルを含む起動処理と比べて，新しくプロセスを起動させる程度の処理で起動が可能であるため，起動時間が短時間で済むという特徴がある．

FastContainer アーキテクチャでは，コンテナが仮想マシンと比較して速く起動できる点と，FastCGI[3] を参考に，物理サーバへの収容効率を高めつつ性能を担保するアーキテクチャを組み合わせる．さらに，HTTP リクエスト毎に負荷状態やアクセス数に応じて，Web アプリケーションコンテナの起動処理，起動継続時間，コンテナの起動数およびリソース割り当てをリアクティブに決定する．一定時間起動することにより，一度コンテナが起動してしまえば，

起動時間に影響なくレスポンスを送信できる．この特性によって，ホストをコンテナ単位で個別に起動しながらも，リクエストの無いコンテナは自然に停止するため，高集積に収容できる．

FastContainer は，コンテナの起動が一般的に高速であること，リクエストを契機としたリアクティブな起動処理であることにより，突発的な負荷に対しても迅速にリクエスト単位でオートスケーリングが可能となる．スケールアップについても，コンテナのリソース管理が cgroup によってプロセス単位で制御されており，cgroup の特徴を利用して，プロセスが処理中であっても CPU 使用時間などの割り当てを即時変更できる．

しかし，このアーキテクチャはコンテナが収容される物理サーバが適切に稼働していることが前提であり，収容サーバが停止すると，手動によるコンテナ再配置が必要であった．そのため，ホスティングサービスやクラウドサービスと同様に，収容サーバが停止した際にサービスを停止させることなく継続的にレスポンスを返すためには，複数の収容サーバにそれぞれ複数インスタンスを配置する必要があり，利用者にとってのサービス利用コストが増加する．また，単一のコンテナで起動している場合は，その収容サーバが停止すると，手動で再配置が行われるまでには同様にサービスも停止してしまう．

3. 提案手法

個人の利用者が WordPress のような一般的な Web コンテンツを配信する仮想化基盤において，収容効率を高めつつ，アクセス集中による負荷やインスタンスの収容サーバの障害に対する可用性を担保可能な基盤にするためには，以下の要件が必要である．

(1) インスタンスで WordPress のような一般的な Web アプリケーションが動作する
(2) 単一のインスタンスであっても収容サーバ障害時には別サーバへ自動的に再配置される
(3) インスタンスの再配置の実行時であっても数秒の遅延で HTTP タイムアウトすることなくオンラインでレスポンスを返せる

本研究では，上述した 3 つの要件を満たすために FastContainer を応用して，HTTP リクエスト単位でコンテナを再配置する仮想化基盤の高速なスケジューリング手法を提案する．提案手法は，HTTP リクエスト処理時において，収容サーバ，および，その経路までの状態に応じて，自動的にコンテナを収容する物理サーバを決定し，サービスを継続させる．FastContainer の状態変化の高速性に基づいて，コンテナが誤検知によって再配置されてもサービスが停止しないことを利用する．それによって，プロキシサーバから収容サーバに ICMP あるいは SO_LINGER オプションを用いた TCP で接続を試みた上で，短いタイムアウトで収容サーバの反応時間を計測できる．実装には，mruby-fast-remote-check[9] を採用した．そのことで，HTTP リクエスト単位でのコンテナスケジューリングを実現する．

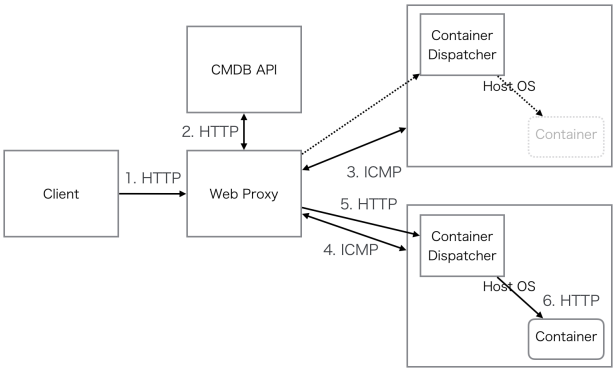

図 1 コンテナスケジューリングフロー
Fig. 1 The Flow of Contaienr Scheduling.

さらに，コンテナ再配置後の最初のリクエストに対する起動時間を速くするために，予めインスタンスの起動処理完了直後のプロセスをイメージ化 (Checkpoint) しておく．起動時は，そのイメージから起動する (Restore) ことにより，通常起動よりも高速化を実現する．

3.1 スケジューリングフロー

図 1 に，コンテナスケジューリングのフローを示す．このフローでは簡単のため，ホストの監視に ICMP を使う．

Host OS が障害を起こしていない正常時のフローについて述べる．図 1 における Client から Container に対して HTTP リクエストがあった場合，Client から Web Proxy に一旦集約される．Web Proxy にリクエストが到達した段階で，Container の収容情報や構成管理情報を保持する CMDB API に情報を問い合せ，Container の収容サーバや IP アドレス/ポート情報を得る．Web Proxy は収容サーバである Host OS 上に存在する Container にリクエストを転送する前に，その Host OS が稼働しているかを ICMP パケットによって確認する．Web Proxy は ICMP の疎通を確認した後に，いったん Host OS 上に稼働している Container Dispatcher にリクエストを転送する．Container Dispatcher はコンテナが稼働していればそのまま HTTP リクエストを転送し，稼働していなければ，再度 CMDB API 情報に基いてコンテナを起動させてからリクエストを転送する．

次に，コンテナが収容されている Host OS が突発的な障害で停止した場合のフローについて述べる．Client から CMDB API を介して Web Proxy に到達し，転送先の収容サーバに ICMP のチェックを行う．その際に，収容サーバがダウンしている場合は，ICMP のチェックタイムアウトを経て，Web Proxy が収容サーバのダウンを認識する．そ

の後，Web Proxy は再度 CMDB API に接続を行い，コンテナの収容情報を他に起動している Host OS の情報に基いて再生成する．その収容情報に基いて，再度起動中の Host OS に ICMP チェックを行い，起動していることを確認した上で，Host OS 上で動作している Container Dispatcher にリクエストを転送する．この場合，収容サーバにはコンテナが起動していないため，Container Dispatcher によって該当コンテナを起動し，HTTP リクエストを転送する．

提案手法では，要件 (1) について，本手法は FastContainer アーキテクチャに基いており，複数の Host OS 間で共有ストレージによりデータを共有しているため，コンテナ上に WordPress のような Web アプリケーションが動作可能であり，かつ，別の Host OS 上でも同様に動作可能である．さらに，要件 (2) について，上記のスケジューリングフローによって，収容サーバがダウンしてもクライアントからの HTTP リクエスト処理の過程で再配置が行われる．要件 (3) を満たすためには，(1) で述べた通り，共有ストレージによってデータを Host OS 間で共有しているため，データのコピーは発生しないことから，ICMP や TCP のタイムアウトの時間と，コンテナの収容サーバ変更後のコンテナ再配置にかかる時間の合計値をできるたけ短くすれば良い．

ICMP や TCP のタイムアウト時間を短くするためには，Web Proxy と Host OS 間の通常の ICMP や TCP に必要とする時間にタイムアウト値をなるべく近くすることが必要となる．同時に，一時的な遅延によって，障害と至らないにも関わらずタイムアウトした場合に再配置が発生したとしても，サービス継続への影響を最小化できれば良い．FastContainer のプロトタイプ実装と実験環境では，単一のコンテナを複数のコンテナにスケールアウトする場合に，レスポンスタイムが短縮されるまでにかかる時間が実験から数秒程度とわかっている [13]．スケールアウト型のスケーリングは，コンテナを追加で起動させる処理であり，コンテナの再配置の起動にかかる処理と同一の処理と言える．そのため，再配置に必要となる時間が論文の環境では数秒程度で完了することが見込める．

3.2 コンテナ環境の起動高速化

再配置時のコンテナの起動時間をさらに短縮するために，コンテナ上で起動するサーバプロセスの起動時間を高速化する．そのために，CRIU を使ってプロセスの起動処理完了直前の状態でプロセスの Checkpoint を行い，プロセスをイメージ化する．その場合，複数の Web サーバソフトウェアに対して，個々に CIRU を組み込むことにより対応していくことは，多数のサーバソフトウェアの拡張が必要となり実装面で非効率である．

任意の GUI アプリケーションに対して，初期化時に実行されるシステムコールを監視しながら，初期化完了直前にプロセスをイメージ化する研究がされている [16]．しかし，任意のシステムコール実行前に自動で停止するといったことができておらず，手動で感覚的に初期化完了時を予想してイメージ化する方が速いといった実験結果になっている．

そこで，個々の Web サーバソフトウェアに機能拡張を実装することなく，汎用的かつ自動でプロセスの Checkpoint/Restore を実現するために，Haconiwa[6] を利用する．Haconiwa は近藤らが開発しているコンテナランタイムであり，コンテナ上で起動するプロセスの完了直後をフックして任意の処理を実行することができる．その機能を利用して，Haconiwa でコンテナ上にプロセスを起動させ，起動完了直後に Haconiwa プロセスから起動処理中のサーバプロセスをイメージ化する．

定期的にプロセスイメージの作成を行い，FastContainer アーキテクチャに従って，コンテナ停止時のリクエストやコンテナの再配置が実行されるタイミングで，作成済みのプロセスイメージから Restore 処理によりプロセスの起動を行う．それによって，起動時間の高速化を実現する．

表 1　実験環境
Table 1　Experimental Environment.

	項目	仕様
Client	ベンチマークを実施するクライアントサーバ	
	CPU	Intel Xeon E5-2650 2.20GHz 1core
	Memory	2GBytes
Compute	コンテナの収容サーバ	
	CPU	Intel Xeon E5-2650 2.20GHz 8core
	Memory	51GBytes
UserDB	コンテナのアプリケーションが利用する DB	
	CPU	Intel Xeon E5-2650 2.20GHz 8core
	Memory	51GBytes
UserProxy	CMDB に基づきコンテナにリクエストを転送	
	CPU	Intel Xeon E5-2650 2.20GHz 1core
	Memory	2GBytes
CoreAPI	コンテナの構成管理情報を制御	
	CPU	Intel Xeon E5-2650 2.20GHz 1core
	Memory	2GBytes
CMDB	コンテナの構成管理情報を保存	
	CPU	Intel Xeon E5-2650 2.20GHz 1core
	Memory	16GBytes
DataPool	コンテナのコンテンツを格納	
	Storage	NetApp FAS8200A
	FlashPool	8.73 TB

4. 実験

本手法の有効性を確認するために，**図 2** に示す FastContainer を用いた実験環境を構築し，コンテナのスケジューリング処理を評価した．**表 1** に実験環境と各種ロールの役割を示す．実験環境の各ロールの NIC と OS は全て，NIC は 1Gbps，OS は Ubuntu16.04，カーネルは 4.4.0-59-generic

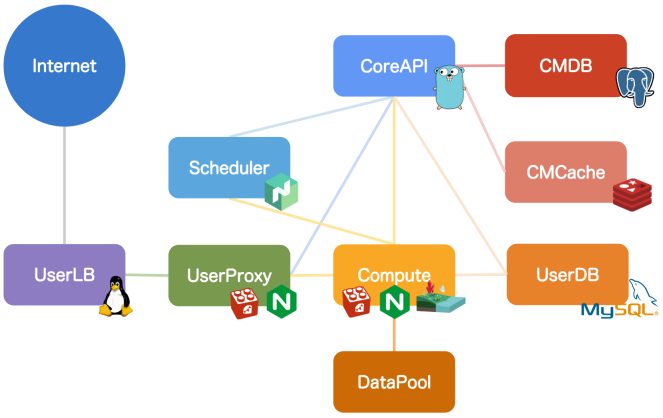

図 2　FastContainer のシステム構成例
Fig. 2　Example of FastContainer System.

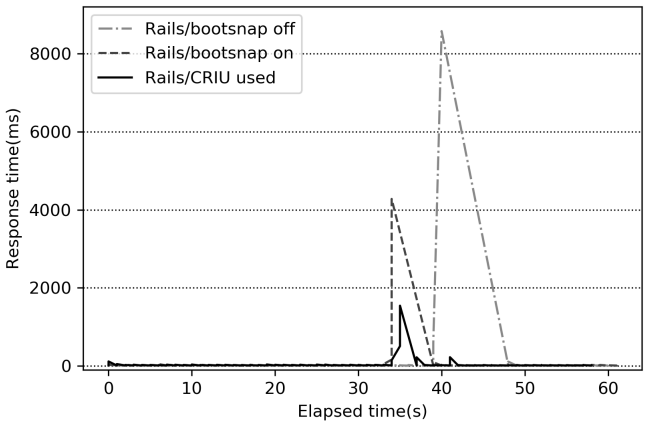

図 4　Ruby on Rails のベンチマーク
Fig. 4　Benchmark of Ruby on Rails.

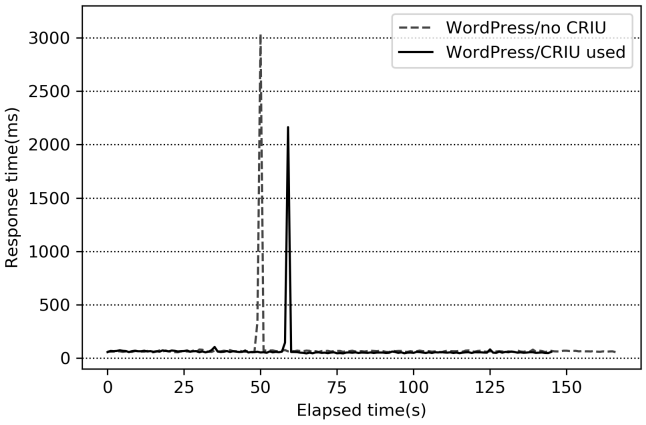

図 3　WordPress のベンチマーク
Fig. 3　Benchmark of WordPress.

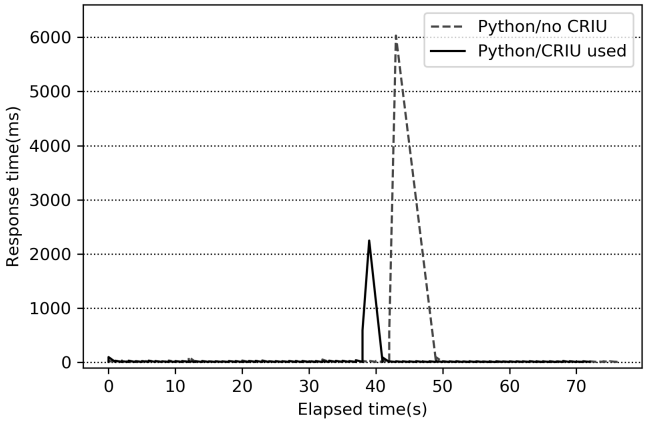

図 5　Django のベンチマーク
Fig. 5　Benchmark of Django.

を利用した．本実験では，Comupte ノードを 2 台利用して，ノードの障害をエミュレートした．コンテナのデータは DataPool 上に保存し，Compute から DataPool に対して NFSv3 でマウントした．

本実験では，Checkpoint/Restore によるサーバプロセスの起動の高速化手法を組み込んだ場合と組み込まない場合とで，Wordpress，Ruby on Rails，Django それぞれのレスポンスタイムを計測した．Wordpress は，Web サーバには Apache 2.4.18，PHP 7.3.0，Wordpress 5.0.3 を利用した．PHP は Opcache 機能を有効にした．また，CRIU によるイメージ化時の Apache のプロセス数は 3，スレッド数 3，単一のプロセスのメモリサイズ (RSS) は 35MBytes であった．

Ruby on Rails は，Ruby 2.5.1，Rails 5.2.1，Puma 3.12.0 を利用し，起動するアプリケーションは現実的な規模（個人/グループ内での利用のアプリケーション程度）で DB を利用したものを採用した[*3]．Ruby on Rails は，gem を事前コンパイルしておくことにより起動を高速化させる bootsnap という機能があるため，CRIU を使わない場合は，bootsnap を使った場合と使わない場合も計測した．また，CRIU によるイメージ化時の Rails のプロセス数は 2，スレッド数は 14，単一のプロセスのメモリサイズ (RSS) は 89MBytes であった．

Django は，Python 3.7.1，Django 2.1.4，gunicorn 19.9.0 を利用し，Rails と同様に，現実的な規模で DB を利用した Web アプリケーションを採用した[*4]．また，CRIU によるイメージ化時の Django のプロセス数は 2，スレッド数は 2，単一のプロセスのメモリサイズ (RSS) は 33MBytes であった．

全ての Web アプリケーションは Haconiwa 0.10.0 で作成したコンテナ上で起動した．データベースは UserDB

*3　https://github.com/everyleaf/el-training
*4　https://mclolipop.zendesk.com/hc/ja/articles/360011824713

上の MySQL 5.7.24 を利用した．コンテナの割り当てメモリは 1024MB であり，CPU は 1 コアを割り当てた．コンテナに対するベンチマークでは，ab コマンドで，同時接続数 1，総接続数 3000 で計測を行い，数十秒たった時点で手動でネットワークを切断し，収容ホストの再配置を発生させた．コンテナの収容サーバに対する TCP 監視には mruby-fast-remote-check を利用して SO_LINGER オプション付きの TCP パケットを送信し，タイムアウトは 100ms とした．

ベンチマーク測定中に，一定時間経過後に Compute 上で ipatables コマンドにより UserProxy からのパケットを drop し，別の Compute にコンテナが再配置されるようにした．今回採用したコンテナランタイムの Haconiwa 0.10.0 の Checkpoint/Restore 機能[*5]を利用して，各 Web アプリケーションをイメージ化しておく．各 Web アプリケーションの起動が完了し、コンテナ内部のポートをリッスンした状態でイメージ化している。したがって、通常起動時はワーカーが全て起動し切っていなくともレスポンスを返すため、CRIU からの起動では若干の条件の違いがある。

ベンチマークでは，秒間のレスポンスタイムの平均値を計算し，時系列データとしてベンチマークが完了するまでのレスポンスタイムの変化を計測した．

図 3 に，WordPress に対するベンチマーク結果を示す．WordPress は，実験環境においては平均的に 70ms 前後のレスポンスタイムであった．45 秒の時点で再配置の処理が動作した際には，3037ms のレスポンスタイムを要した．TCP のタイムアウト値が 100ms であることから，コンテナの再配置から Apache の起動とレスポンス生成までには概ね 2937ms で実現できている．そのことから，コンテナの再配置とコンテナ起動に要する時間は，WordPress のレスポンスタイム 70ms を除くと，2867ms となる．

一方，CRIU のイメージ化を使った場合の Wordpress に対するベンチマークは，ベンチマークが 55 秒経過したときに，手動でネットワークを切断した．WordPress は，CRIU を使わない場合の実験と同様に，平均的に 70ms 前後のレスポンスタイムであった．55 秒の時点で再配置の処理が動作した際には，2163ms の処理時間を要した．TCP のタイムアウト値が 100ms であることから，コンテナの再配置から Apache の起動とレスポンス生成までには概ね 2063ms で実現できている．そのことから，コンテナの再配置とコンテナ起動に要する時間は，WordPress のレスポンスタイム 70ms を除くと，1993ms となる．これらの結果から，CRIU を使ったイメージ化を組み合わせることにより，874ms 高速に起動できた．高速に起動できることから，収容サーバ障害時にも，874ms 速くレスポンスを返すことができる．

[*5] https://github.com/haconiwa/haconiwa/pull/188

図 4 に，Ruby on Rails に対するベンチマーク結果を示す．Ruby on Rails は，実験環境においては平均的に 20ms 前後のレスポンスタイムであった．CRIU も bootsnap も使わない場合は，38 秒の時点で再配置の処理が発生した際に，8575ms のレスポンスタイムを要した．TCP のタイムアウト値が 100ms であることから，コンテナの再配置から Rails の起動とレスポンス生成までには概ね 8475ms で実現できている．そのことから，コンテナの再配置とコンテナ起動に要する時間は，Rails のレスポンスタイム 70ms を除くと，8455ms となる．

一方，CRIU のイメージ化を使った場合は，35 秒の時点で再配置の処理が動作した際には，1544ms の処理時間であった．TCP のタイムアウト値が 100ms であることから，コンテナの再配置から Rails の起動とレスポンス生成までには概ね 1444ms で実現できている．そのことから，コンテナの再配置とコンテナ起動に要する時間は，レスポンスタイム 20ms を除くと，1424ms となる．これらの結果から，CRIU を使ったイメージ化を組み合わせることにより，7011ms 高速に起動できた．同様に，bootsnap と比較した場合は，同様の計算式から，2734ms 高速に起動できた．

図 5 に，Django に対するベンチマーク結果を示す．Django は，実験環境においては平均的に 15ms 前後のレスポンスタイムであった．42 秒の時点で再配置の処理が動作した際に，6030ms のレスポンスタイムを要した．Wordpress や Rails のレスポンスタイムと同様の計算式から，コンテナの再配置とコンテナ，および，Web アプリケーション起動に要する時間は，5915ms となる．同様に，CRIU を使った場合は，コンテナの再配置実行時に 2249ms のレスポンスタイムになっている．つまり，CRIU を使ったイメージ化を組み合わせることにより，3781ms 高速に再配置とレスポンス送信が可能になる．

UserProxy から Compute にリクエスト転送が送信されて，Compute 上のコンテナから UserProxy にレスポンスデータが送信される前の状態で Compute が停止してしまった場合は，そのレスポンスを UserProxy が正しく受信できないため，レスポンスを一部取りこぼすこともありえる．しかし，これは従来方式において複数のホスト OS にそれぞれ分散してインスタンスを配置して可用性を担保する方式でも原理的に生じるため，本手法の特徴的な問題ではない．

本実験結果から，単一のコンテナインスタンスを単一の収容サーバに起動させておきながらも，その収容サーバの停止時には，プロセスのイメージ化を組み合わせることにより，迅速に再配置のスケジューリングが行えていることがわかった．また，実験環境程度のハードウェアスペックで，2 秒程度であれば，実用上は HTTP タイムアウトすることなくレスポンスを返せると見込める．Ruby on Rails や Django のように，起動してしまえばレスポンスは速い

が，起動に時間がかかるような Web アプリケーションに対して，大きな効果が見込めることもわかった．一方で，コンテナの再配置が頻発する場合には，提供サービスレベルにも依存する問題でもあるため，再配置の頻度についてはサービスに合わせた検討が必要である．

5. まとめ

本研究では，HTTP リクエスト処理時において，収容サーバ，および，その経路までの状態に応じて，自動的にコンテナを収容する物理サーバを決定し，サービスを継続させる，HTTP リクエスト単位でのコンテナスケジューリング手法を提案し，その有効性を示した．

実験から，一般的な Web アプリケーションが起動しているコンテナにおいて，複数のホスト OS に複数のインスタンスを配置して可用性を担保することなく，単一のインスタンスという少ないリソースで，1.5s から 2s 程度の再配置処理により，HTTP エラーを発生させることなく可用性を担保することができた．このようなリソース使用量の削減により，サービス利用者は可用性の担保のために，複数の収容サーバに横断的に複数のインスタンスを配置する必要がなくなり，サービス利用料を削減できる．また，データセンターやホスト OS を抽象化して，障害や高負荷時に自律的に収容サーバを移動するようなスケジューリングが可能となる．

今後の課題として，コンテナ起動時間の高速化手法をプロダクション環境で評価を行うことや，負荷やレスポンスタイムに応じた HTTP リクエスト単位でのより適応的なコンテナスケジューリング手法の実現が挙げられる．

参考文献

[1] Amazon Web Services: Lambda, https://aws.amazon.com/lambda/.
[2] CRIU – A project to implement checkpoint/restore functionality for Linux, https://criu.org/.
[3] Brown Mark R, FastCGI: A high-performance gateway interface, Fifth International World Wide Web Conference. Vol. 6. 1996.
[4] Felter W, Ferreira A, Rajamony R, Rubio J, An Updated Performance Comparison of Virtual Machines and Linux Containers, IEEE International Symposium Performance Analysis of Systems and Software (ISPASS), pp. 171-172, March 2015.
[5] Ferdman M, Adileh A, Kocberber O, Volos S, Alisafaee M, Jevdjic D, Falsafi B, Clearing the clouds: a study of emerging scale-out workloads on modern hardware, ACM SIGPLAN Notices, Vol. 47, No. 4, pp. 37-48, March 2012.
[6] Haconiwa - the mruby on container, https://haconiwa.mruby.org/.
[7] P Mell, T Grance, The NIST Definition of Cloud Computing", US Nat'l Inst. of Science and Technology, 2011, http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.
[8] Prodan R, Ostermann S, A Survey and Taxonomy of Infrastructure as a Service and Web Hosting Cloud Providers,10th IEEE/ACM International Conference on Grid Computing, pp. 17-25, October 2009.
[9] Ryosuke Matsumoto, mruby-fast-remote-check, https://github.com/matsumotory/mruby-fast-remote-check.
[10] 笠原 義晃, 松本 亮介, 近藤 宇智朗, 小田 知央, 嶋吉 隆夫, 金子晃介, 岡村 耕二, 軽量コンテナに基づく柔軟なホスティング・クラウド基盤の研究開発と大規模・高負荷テスト環境の構築, 研究報告インターネットと運用技術（IOT）, Vol.2018-IOT-40(2), pp.1-8, 2018 年 3 月.
[11] 松本 亮介, Web サーバの高集積マルチテナントアーキテクチャに関する研究, 京都大学博士 (情報学) 学位論文, 2017.
[12] 松本亮介, 川原将司, 松岡輝夫, 大規模共有型 Web バーチャルホスティング基盤のセキュリティと運用技術の改善, 情報処理学会論文誌, Vol.54, No.3, pp.1077-1086, 2013 年 3 月.
[13] 松本亮介, 近藤宇智朗, 三宅悠介, 力武健次, 栗林健太郎, FastContainer: 実行環境の変化に素早く適応できる恒常性を持つシステムアーキテクチャ, インターネットと運用技術シンポジウム 2017 論文集, 2017, 89-97（2017-11-30）, 2017 年 12 月.
[14] 松本亮介, 栗林 健太郎, 岡部寿男, リクエスト単位で仮想的にハードウェアリソースを分離する Web サーバのリソース制御アーキテクチャ, 情報処理学会論文誌, Vol.59, No.3, pp.1016-1025, 2018 年 3 月.
[15] 三宅 悠介, 松本 亮介, 力武 健次, 栗林 健太郎, アクセス頻度予測に基づく仮想サーバの計画的オートスケーリング, 情報処理学会研究報告インターネットと運用技術（IOT）,2017-IOT-38, Vol.13, pp.1-8, 2017 年 6 月.
[16] 阿部 敏和, 中山 泰一, プロセスの保存と復元による GUI アプリケーションの起動高速化, 情報処理学会全国大会講演論文集, 第 72 回アーキテクチャ, pp95-96, 2010 年 3 月.